Atty Docket No. 005313.00001

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent of:

Marc D. Van Heyningen

U.S. Pat. App. No.: 09/782,593

Filing Date: February 12, 2001

Examiner: L. Son

Group Art Unit: 2135

For: METHOD AND APPARATUS FOR PROVIDING SECURE STREAMING DATA TRANSMISSION FACILITIES USING UNRELIABLE PROTOCOLS

DECLARATION UNDER 37 C.F.R. § 1.131

Commissioner for Patents
P.O. Box 1450,
Alexandria, Virginia 22313-1450

Sir:

I, Marc D. Van Heyningen, do hereby declare as follows:

1. I am named as the inventor in U.S. Patent Application No. 09/782,593, filed February 12, 2001. I am an employee of Aventail Corporation having a place of business at 808 Howell Street, Second Floor, Seattle, Washington 98101, and I am over the age of eighteen years.

2. I am advised that Exhibit A contains a true and correct copy of the claims currently pending in U.S. Patent Application No. 09/782,593, namely claims 1-22. I have read and understand these claims.

3. I also have been informed that the United States Patent and Trademark Office has rejected claims 4-7 and 10-22 as contained in Exhibit A based upon U.S. Patent Application No. 2002/0094085 A1 to Roberts. I understand that the Roberts patent claims an effective U.S. filing

date of January 16, 2001.

4. I have reviewed (a) the computer code file entitled "ssletx.c" (a printed copy of which is attached as Exhibit B), (b) the computer code file entitled "sslrec.c" (a printed copy of which is attached as Exhibit C), (c) the computer code file entitled "ciphers.c" (a printed copy of which is attached as Exhibit D), and (d) the computer code entitled "sslenv.c" (a printed copy of which is attached as Exhibit E). I created each of these files of computer code by revising existing computer code to implement the invention as described in my patent application and recited in claims 4-7 and 10-22 (Exhibit A). I have obtained each of these files of computer code from our company's records. The copy of each of these files of computer code has been changed to include line numbers for the code lines. The line numbers were added in Exhibit B through Exhibit E to make the discussion below easier to follow, i.e., so that the cited code line numbers below would be readily available with the attached copy of the code.

5. As will be described in more detail below, the attached computer code, when implemented on a computer system, allows one to encrypt and transmit data records between a first computer and a second computer using an unreliable communication protocol in the manner described in my patent application and recited in claims 4-7 and 10-22 (Exhibit A).

6. One feature of a claim in my patent application relates to encrypting and transmitting data records between a first computer and a second computer using an unreliable communication protocol wherein each data record is encrypted by incorporating a nonce and without reference to a previously transmitted data record (note, for example, claims 4-6).

7. Referencing the source code, portions of the code that implement functionality relating

to my invention typically reference the term "SSLoppy", the working name for the development of the invention when this code was created. This term refers to the features provided by the invention that allows Secure Sockets Layer (SSL) records to be dropped and reordered during communication without disruption, which thus offer a "sloppy" variation of the SSL communication technique. The role of the first computer is included in: sslrec.c, lines 326-336, where the computer code adds extra room to a record for the inclusion of an explicit nonce, lines 354-378, where the computer code generates the nonce randomly, uses it in place of the sequence number in computing the MAC, and includes it in the record being built, lines 406-410, where the computer code passes this nonce to the encryption function, and in file cipher.c, lines 252-257, where the computer code uses the passed nonce value as an initial vector (IV), if present. The role of the second computer in reading these records is described in: file sslrec.c, lines 450-455, where the code parses the nonce out of incoming records, lines 466-475, where the code passes this nonce to the decryption function, and in file cipher.c, lines 323-328, where the code uses the passed nonce as the initial vector (IV), if present.

8. Another feature of a claim in my patent application relates to encrypting and transmitting data records between a first computer and a second computer using an unreliable communication protocol wherein an indicator is embedded in each of the data records indicating that the data records are encrypted according to an encryption scheme that encrypts records without regard to any previously transmitted data records, and the second computer determines whether the indicator is present in each record and, in response to determining that the indicator is not present, processes each such record differently than if the indicator is set (note, for example, claim 7).

9. Referencing the source code, in file sslrec.c, the role of the first computer in sending its different form of data is included in: lines 326-336, where the software code allocates extra space in the record for the nonce, if necessary, lines 354-391, where the ssloppy flag is used to decide whether to generate the record differently, lines 406-410, where the same flag is used to decide whether to use the nonce as the initial vector (IV) for the symmetric cipher algorithm, and lines 424-428, where the flag is used to decide whether to increment the sequence number. The role of the second computer, in receiving a packet of a different form of data, is included in: lines 95-99, where the software code checks a flag in the incoming record to verify that it is only receiving the new type of reorderable data when expected, lines 221-222, where the software code uses the same flag to determine whether to increment the sequence number, lines 450-454, where the software code uses the same flag to determine whether to extract the nonce from the record, lines 466-480, where the software code uses the same flag to decide whether to use this nonce as the initial vector (IV) when decrypting the record, and lines 497-517, where the software code uses the same flag to decide whether to use the nonce in place of the sequence number when verifying the MAC.

10. Still another feature of a claim in my patent application relates to securely transmitting a plurality of data records between a client computer and a proxy server using an unreliable communication protocol, that include the steps of establishing a reliable connection between the client computer and the proxy server, exchanging encryption credentials between the client computer and the proxy server over the reliable connection generating a nonce for each of a plurality of data records, wherein each nonce comprises an initialization vector necessary to

decrypt a corresponding one of the plurality of data records, using the nonce to encrypt each of the plurality of data records and appending the nonce to each of the plurality of data records, transmitting the plurality of encrypted data records from the client computer to the proxy server using an unreliable communication protocol, and, in the proxy server, decrypting each of the plurality of encrypted data records using a corresponding nonce extracted from each data record and a previously shared encryption key (note claims 10-15).

11. The proxy server and client architecture described above was already a commercial shipping product from Aventail, Inc. by January 16, 2001. Prior to the enhancements offered by the implementation of the invention, the product utilized the unreliable communication protocol "User Data Protocol" (UDP) to relay datagram records over a communication medium without any cryptographic protection. Comments in the software code refer to this process as "UDP Naked." The addition of the invention replaced "UDP Naked" with the "SSLoppy" process (as described above) thereby allowing communications to be cryptographically protected. Referencing the file sslenv.c, lines 1056-1067, for transmitting a datagram with this SSLoppy feature enabled, the SSLoppy encryption feature is turned on for the processing of a record by calling SSLSetSloppyMode to 1, and, subsequent to processing of the record, resetting this value back to 0 (described in lines 1131, 1145, 1153, 1183, 1220, 1244, 1252, 1259, 1278, 1286, 1314, 1347, 1359, and 1397). This software code processes only designated datagrams using the SSLoppy encryption process, and provides for standard SSL processing of data records being exchanged via a reliable communication protocol. The underlying functionality for selecting the SSLoppy encryption process is in sslctx.c, lines 899-911, where this function call sets the ctx-

>ssloppy flag. This flag is used to read and write records in the mode documented in paragraph 9 above.

12. Still another feature of a claim in my patent application relates to a system for securely transmitting data using an unreliable protocol that includes a first computer comprising a communication protocol client function operable in conjunction with an application program to transmit data records securely using an unreliable protocol, and a second computer coupled to the first computer and comprising a communication protocol server function operable in conjunction with the communication protocol client function to receive data records securely using the unreliable communication protocol, wherein the communication protocol client function encrypts each data record using a nonce and an encryption key and appends the respective nonce to each of the encrypted data records, and wherein the communication protocol server function decrypts each of the data records using the respectively appended nonce and the encryption key (note claims 16-22).

13. Supporting code is the same as that from paragraph 11 above.

14. The "source code" software code existed in the form shown in Exhibits B through E prior to January 16, 2001. This computer code received unit testing and was used as a proof-of-concept in the development process. Using this software code, the invention performed in its intended manner prior to January 16, 2001, as described in our patent application and claims 4-7 and 10-22 (as shown in Exhibit A), and thus was actually reduced to practice prior to January 16, 2001.

U.S. Pat. App. No.: 09/782,593
Atty. Docket No.: 005313.00001

DECLARATION IN LIEU OF OATH

15. I further declare that all information stated herein based upon my own knowledge is true and that all information stated herein based on information and belief is believed to be true, and further that the statements made in this Declaration were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of this application or any patent issuing based on this application.

Date:

By: _____
Marc Van Heyningen, Inventor

EXHIBIT A

1. A method of transmitting data securely over a computer network, comprising the steps of:

(1) establishing a communication path between a first computer and a second computer;

(2) encrypting and transmitting data records between the first computer and the second computer using an unreliable communication protocol, wherein each data record is encrypted without reference to a previously transmitted data record; and

(3) in the second computer, receiving and decrypting the data records transmitted in step (2) without reference to a previously received data record.

2. The method of claim 1, further comprising the step of, prior to step (1), establishing a reliable communication path between the first computer and the second computer and exchanging security credentials over the reliable communication path.

3. The method of claim 2, wherein the step of exchanging security credentials comprises the step of exchanging an encryption key that is used to encrypt the data records in step (2).

4. The method of claim 1, wherein step (2) comprises the step of incorporating a nonce in each data record that is used by the second computer in combination with a previously shared encryption key to decrypt each of the data records in step (3).

5. The method of claim 4, wherein the nonce comprises a random number.

6. The method of claim 4, further comprising the step of, in the second computer, verifying that the nonce has not previously been received in a previously transmitted data record.

7. The method of claim 1,

wherein step (2) comprises the step of embedding an indicator in each of the data records indicating that the data records are encrypted according to an encryption scheme that encrypts records without regard to any previously transmitted data records, and

wherein step (3) comprises the step of determining whether the indicator is present in each record and, in response to determining that the indicator is not present, processing each such record differently than if the indicator is set.

8. The method of claim 1, wherein step (1) is performed using the Transmission Control Protocol, and wherein step (2) is performed using the User Datagram Protocol.

9. The method of claim 1, wherein step (2) is performed by a proxy server that encrypts data records received from another server.

10. A method of securely transmitting a plurality of data records between a client computer and a proxy server using an unreliable communication protocol, comprising the steps of:

(1) establishing a reliable connection between the client computer and the proxy server;

(2) exchanging encryption credentials between the client computer and the proxy server over the reliable connection;

(3) generating a nonce for each of a plurality of data records, wherein each nonce comprises an initialization vector necessary to decrypt a corresponding one of the plurality of data records;

(4) using the nonce to encrypt each of the plurality of data records and appending the nonce to each of the plurality of data records;

(5) transmitting the plurality of data records encrypted in step (4) from the client computer to the proxy server using an unreliable communication protocol; and

(6) in the proxy server, decrypting each of the plurality of encrypted data records using a corresponding nonce extracted from each data record and a previously shared encryption key.

11. The method of claim 10, wherein step (6) comprises the step of checking to determine whether each data record received from the client computer is formatted according to a secure unreliable transmission format and, if a particular record is not formatted according to a secure unreliable transmission format, bypassing the decryption using the corresponding nonce.

12. The method of claim 10, wherein step (3) comprises the step of generating a random number as each nonce.

13. The method of claim 10, wherein step (1) is performed using Transmission Control Protocol, and wherein step (5) is performed using User Datagram Protocol.

14. The method of claim 10, wherein step (6) is performed using an encryption key previously shared using a reliable communication protocol.

15. The method of claim 14, wherein the reliable communication protocol is Transmission Control Protocol.

16. A system for securely transmitting data using an unreliable protocol, comprising:

a first computer comprising a communication protocol client function operable in conjunction with an application program to transmit data records securely using an unreliable protocol; and

a second computer coupled to the first computer and comprising a communication protocol server function operable in conjunction with the communication protocol client function to receive data records securely using the unreliable communication protocol,

wherein the communication protocol client function encrypts each data record using a nonce and an encryption key and appends the respective nonce to each of the encrypted data records; and

wherein the communication protocol server function decrypts each of the data records using the respectively appended nonce and the encryption key.

17. The system of claim 16, wherein the communication protocol client function exchanges encryption credentials with the communication protocol server function using a reliable communication protocol.

18. The system of claim 17, wherein the unreliable communication protocol comprises the User Datagram Protocol, and wherein the reliable communication protocol comprises the Transmission Control Protocol.

19. The system of claim 16, wherein the communication protocol client function and the communication protocol server function are compatible with the SOCKS communication protocol.

20. The system of claim 16, wherein the communication protocol client function and the communication protocol server function are compatible with the SSL/TLS communication protocol.

21. The system of claim 16, wherein the second computer comprises a proxy server that forwards decrypted records received from the first computer to a server computer.

22. The system of claim 16, wherein the second computer comprises a record detector that determines whether an indicator has been set in each data record received from the first computer and, if the indicator has not been set, bypassing decryption in the server computer.

EXHIBIT B

```
 0 #include <stdio.h>
 1
 2 /* #define HYPER_DEBUG 1 */
 3
 4 /*  ********************************************************************
 5     File: sslctx.c
 6
 7     SSL Plus: Security Integration Suite(tm)
 8     Version 1.1.1 -- August 11, 1997
 9
10     Copyright (c)1996, 1997 by Consensus Development Corporation
11          Copyright (c)1997, 1998 by Aventail Corporation
12
13     Portions of this software are based on SSLRef(tm) 3.0, which is
14     Copyright (c)1996 by Netscape Communications Corporation. SSLRef(tm)
15     was developed by Netscape Communications Corporation and Consensus
16     Development Corporation.
17
18     In order to obtain this software, your company must have signed
19     either a PRODUCT EVALUATION LICENSE (a copy of which is included in
20     the file "LICENSE.TXT"), or a PRODUCT DEVELOPMENT LICENSE. These
21     licenses have different limitations regarding how you are allowed to
22     use the software. Before retrieving (or using) this software, you
23     *must* ascertain which of these licenses your company currently
24     holds. Then, by retrieving (or using) this software you agree to
25     abide by the particular terms of that license. If you do not agree
26     to abide by the particular terms of that license, than you must
27     immediately delete this software. If your company does not have a
28     signed license of either kind, then you must either contact
29     Consensus Development and execute a valid license before retrieving
30     (or using) this software, or immediately delete this software.
31
32     ********************************************************************
33
34     File: sslctx.c     SSLContext accessors
35
36     Functions called by the end user which configure an SSLContext
37     structure or access data stored there.
38
```

<because of the size of this file and the fact that most of it is not relevant, we include only the relevant function SSLSetSloppyMode here>

```
899 SSLErr CDECL SSLSetSloppyMode(SSLContext *ctx, int m)
900 {
901    if(ctx == NULL)
902            return SSLUnknownErr;
903
904    if((ctx->selectedCipherSpec->cipher->blockSize == 0) &&
905       (ctx->selectedCipherSpec->cipher->keySize > 0))
906            return SSLProtocolErr;
907
908    ctx->ssloppy = m;
909
910    return SSLNoErr;
911 }
```

EXHIBIT C

```
0 /*  ********************************************************************
1     File: sslrec.c
2
3     SSL Plus: Security Integration Suite(tm)
4     Version 1.1.1 -- August 11, 1997
5
6     Copyright (c)1996, 1997 by Consensus Development Corporation
7           Copyright (c)1997, 1998 by Aventail Corporation
8
9     Portions of this software are based on SSLRef(tm) 3.0, which is
10    Copyright (c)1996 by Netscape Communications Corporation. SSLRef(tm)
11    was developed by Netscape Communications Corporation and Consensus
12    Development Corporation.
13
14    In order to obtain this software, your company must have signed
15    either a PRODUCT EVALUATION LICENSE (a copy of which is included in
16    the file "LICENSE.TXT"), or a PRODUCT DEVELOPMENT LICENSE. These
17    licenses have different limitations regarding how you are allowed to
18    use the software. Before retrieving (or using) this software, you
19    *must* ascertain which of these licenses your company currently
20    holds. Then, by retrieving (or using) this software you agree to
21    abide by the particular terms of that license. If you do not agree
22    to abide by the particular terms of that license, than you must
23    immediately delete this software. If your company does not have a
24    signed license of either kind, then you must either contact
25    Consensus Development and execute a valid license before retrieving
26    (or using) this software, or immediately delete this software.
27
28    ********************************************************************
29
30    File: sslrec.c     Encryption, decryption and MACing of data
31
32    All the transformations which occur between plaintext and the
33    secured, authenticated data that goes out over the wire. Also,
34    detects incoming SSL 2 hello messages and hands them off to the SSL 2
35    record layer (and hands all SSL 2 reading & writing off to the SSL 2
36    layer).
37
38    ******************************************************************** */
39
40 /* #define HYPER_DEBUG 1 */
41
42 #ifdef HYPER_DEBUG
43 #include <stdio.h>
44 #endif
45
46 #ifndef _SSL_H_
47 #include "ssl.h"
48 #endif
49
50 #ifndef _SSLREC_H_
51 #include "sslrec.h"
52 #endif
53
54 #ifndef _SSLALLOC_H_
55 #include "sslalloc.h"
56 #endif
57
58 #ifndef _CRYPTYPE_H_
59 #include "cryptype.h"
60 #endif
61
62 #ifndef _SSLCTX_H_
63 #include "sslctx.h"
64 #endif
65
66 #ifndef _SSLALERT_H_
67 #include "sslalert.h"
68 #endif
69
70 #ifndef _SSL2_H_
```

```
#include "ssl2.h"
#endif

#include <string.h>

static SSLErr DecryptSSLRecord(uint8 type, SSLBuffer *payload, SSLContext *ctx);
static SSLErr VerifyMAC(uint8 type, SSLBuffer data, uint8 *compareMAC, uint64 seqNo, SSLContext
  *ctx);
static SSLErr ComputeMAC(uint8 type, SSLBuffer data, SSLBuffer mac, uint64 seqNo, SSLBuffer
  secret, CipherContext *cipherCtx, SSLContext *ctx);
static uint8* SSLEncodeUInt64(uint8 *p, uint64 value);

/* ReadSSLRecord
 *  Attempt to read & decrypt an SSL record.
 */
SSLErr
SSLReadRecord(SSLRecord *rec, SSLContext *ctx)
{   SSLErr      err;
    uint32      len, contentLen;
    uint8           *progress;
    SSLBuffer       readData, cipherFragment;

#ifdef HYPER_DEBUG
  fprintf(stderr, "Got into SSLReadRecord, whee!\n");
#endif

  /* if we get UDP data when we aren't expecting it, that's really bad,
    so report an appropriate error. */
  if((rec->contentType == SSL_application_data_ssloppy) &&
     (! ctx->ssloppy))
         return SSLProtocolErr;


  if (!ctx->partialReadBuffer.data || ctx->partialReadBuffer.length < 5)
   {   if (ctx->partialReadBuffer.data)
           if ((err = SSLFreeBuffer(&ctx->partialReadBuffer, &ctx->sysCtx)) != 0)
           {   SSLFatalSessionAlert(alert_close_notify, ctx);
               return ERR(err);
           }
       if ((err = SSLAllocBuffer(&ctx->partialReadBuffer, DEFAULT_BUFFER_SIZE, &ctx->sysCtx))
    != 0)
       {   SSLFatalSessionAlert(alert_close_notify, ctx);
           return ERR(err);
       }
   }

   if (ctx->protocolVersion == SSL_Version_Undetermined ||
       ctx->protocolVersion == SSL_Version_3_0_With_2_0_Hello)
       if (ctx->amountRead < 1)
       {   readData.length = 1 - ctx->amountRead;
           readData.data = ctx->partialReadBuffer.data + ctx->amountRead;
           len = readData.length;
           if (ERR(err = ctx->ioCtx.read(readData, &len, ctx->ioCtx.ioRef)) != 0)
           {   if (err == SSLWouldBlockErr)
                   ctx->amountRead += len;
               else
                   SSLFatalSessionAlert(alert_close_notify, ctx);
               return err;
           }
           ctx->amountRead += len;
       }

/* In undetermined cases, if the first byte isn't in the range of SSL 3.0
 *  record types, this is an SSL 2.0 record
 */
   switch (ctx->protocolVersion)
   {   case SSL_Version_Undetermined:
       case SSL_Version_3_0_With_2_0_Hello:
           if (ctx->partialReadBuffer.data[0] < SSL_smallest_3_0_type ||
               ctx->partialReadBuffer.data[0] > SSL_largest_3_0_type)
```

```
139                 return SSL2ReadRecord(rec, ctx);
140             else
141                 break;
142         case SSL_Version_2_0:
143             return SSL2ReadRecord(rec, ctx);
144         default:
145             break;
146     }
147
148
149 #ifdef HYPER_DEBUG
150   fprintf(stderr, "About to get into the read callback stuff\n");
151 #endif
152     if (ctx->amountRead < 5)
153     {   readData.length = 5 - ctx->amountRead;
154         readData.data = ctx->partialReadBuffer.data + ctx->amountRead;
155         len = readData.length;
156         if (ERR(err = ctx->ioCtx.read(readData, &len, ctx->ioCtx.ioRef)) != 0)
157         {   if (err == SSLWouldBlockErr)
158                 ctx->amountRead += len;
159                         else if (err == SSLIOClosedOverrideGoodbyeKiss && ctx->amountRead ==
    0)
160                         {   SSLClose(ctx);
161                         return SSLConnectionClosedGraceful;
162                         }
163                         else
164                                     SSLFatalSessionAlert(alert_close_notify, ctx);
165                         return err;
166                         }
167         ctx->amountRead += len;
168     }
169
170     ASSERT(ctx->amountRead >= 5);
171
172     progress = ctx->partialReadBuffer.data;
173     rec->contentType = *progress++;
174     if (rec->contentType < SSL_smallest_3_0_type ||
175         rec->contentType > SSL_largest_3_0_type)
176         return ERR(SSLProtocolErr);
177
178     rec->protocolVersion = (SSLProtocolVersion)SSLDecodeInt(progress, 2);
179     progress += 2;
180     contentLen = SSLDecodeInt(progress, 2);
181     progress += 2;
182     if (contentLen > (16384 + 2048))    /* Maximum legal length of an SSLCipherText payload */
183     {   SSLFatalSessionAlert(alert_unexpected_message, ctx);
184         return ERR(SSLProtocolErr);
185     }
186
187     if (ctx->partialReadBuffer.length < 5 + contentLen)
188     {   if ((err = SSLReallocBuffer(&ctx->partialReadBuffer, 5 + contentLen, &ctx->sysCtx)) !=
    0)
189         {   SSLFatalSessionAlert(alert_close_notify, ctx);
190             return ERR(err);
191         }
192     }
193
194     if (ctx->amountRead < 5 + contentLen)
195     {   readData.length = 5 + contentLen - ctx->amountRead;
196         readData.data = ctx->partialReadBuffer.data + ctx->amountRead;
197         len = readData.length;
198         if (ERR(err = ctx->ioCtx.read(readData, &len, ctx->ioCtx.ioRef)) != 0)
199         {   if (err == SSLWouldBlockErr)
200                 ctx->amountRead += len;
201             else
202                 SSLFatalSessionAlert(alert_close_notify, ctx);
203             return err;
204         }
205         ctx->amountRead += len;
206     }
207
```

```
208     ASSERT(ctx->amountRead >= 5 + contentLen);
209
210     cipherFragment.data = ctx->partialReadBuffer.data + 5;
211     cipherFragment.length = contentLen;
212
213 /* Decrypt the payload & check the MAC, modifying the length of the buffer to indicate the
214  *  amount of plaintext data after adjusting for the block size and removing the MAC
215  *  (this function generates its own alerts)
216  */
217     if ((err = DecryptSSLRecord(rec->contentType, &cipherFragment, ctx)) != 0)
218         return err;
219
220 /* We appear to have sucessfully received a record; increment the sequence number */
221     if(rec->contentType != SSL_application_data_ssloppy)
222         IncrementUInt64(&ctx->readCipher.sequenceNum);
223
224
225 #ifdef SSL_COMPRESSION
226         if((ctx->compressNow) && (ctx->selectedCompression != NULL) &&
227                 (ctx->selectedCompression->identifier != 0)) {
228
229 /* Allocate a buffer to return the plaintext in and return it */
230                         if ((err = SSLAllocBuffer(&rec->contents, DEFAULT_BUFFER_SIZE,
231
232                                 &ctx->sysCtx)) != SSLNoErr) {
233                                         SSLFatalSessionAlert(alert_close_notify, ctx);
234                                         return ERR(err);
235                         )
236                         if((err = ctx->selectedCompression->process(cipherFragment,
237
238             &(rec->contents),
239
240             ctx->readCompressRef,
241
242             ctx)) != SSLNoErr) {
243                                         SSLFreeBuffer(&rec->contents, &ctx->sysCtx);
244                                         SSLFatalSessionAlert(alert_decompression_failure, ctx);
245                                         return ERR(err);
246                         }
247 #ifdef HYPER_DEBUG
248                         fprintf(stderr, "Deompression created output of %d from size %d\n",
249                                                         rec->contents.length,
    cipherFragment.length);
250 #endif
251         } else {
252                         if ((err = SSLAllocBuffer(&rec->contents, cipherFragment.length,
253
254                                 &ctx->sysCtx)) != 0)
255                         {
256                                         SSLFatalSessionAlert(alert_close_notify, ctx);
257                                         return ERR(err);
258                         }
259                         memcpy(rec->contents.data, cipherFragment.data, (size_t)
    cipherFragment.length);
260         }
261 #else
262         memcpy(rec->contents.data, cipherFragment.data, (size_t) cipherFragment.length);
263 #endif
264
265     ctx->amountRead = 0;        /* We've used all the data in the cache */
266
267     return SSLNoErr;
268 }
269
270 /* SSLWriteRecord does not send alerts on failure, out of the assumption/fear
271  *  that this might result in a loop (since sending an alert causes SSLWriteRecord
272  *  to be called).
273  */
```

```
SSLErr
SSLWriteRecord(SSLRecord rec, SSLContext *ctx)
{   SSLErr      err;
    int         padding = 0, i, freerec = 0;
    WaitingRecord   *out, *queue;
    SSLBuffer       buf, payload, secret, mac, nonce;
    uint8           *progress;
    uint16      payloadSize,blockSize,nonceSize = 0;

    if (rec.protocolVersion == SSL_Version_2_0)
        return SSL2WriteRecord(rec, ctx);

    ASSERT(rec.protocolVersion == SSL_Version_3_0);
    ASSERT(rec.contents.length <= 16384);

#ifdef SSL_COMPRESSION
        if((ctx->compressNow) && (ctx->selectedCompression != NULL) &&
                (ctx->selectedCompression->identifier != 0)) {
                    SSLBuffer compdata;

                    /* make a guess about how long the buffer will need to be */
                    if((err = SSLAllocBuffer(&compdata, rec.contents.length + 4,

                     &ctx->sysCtx)) != SSLNoErr)
                                            return ERR(err);
                    if((err = ctx->selectedCompression->process(rec.contents, &compdata,

        ctx->writeCompressRef,

        ctx)) != SSLNoErr) {
                                        SSLFreeBuffer(&compdata, &ctx->sysCtx);
                                        return ERR(err);
                    )

                    rec.contents = compdata;
                    freerec = 1;
        )
#endif

    out = 0;
    /* Allocate a WaitingRecord to store our ready-to-send record in */
    if ((err = SSLAllocBuffer(&buf, sizeof(WaitingRecord), &ctx->sysCtx)) != 0)
        return ERR(err);
    out = (WaitingRecord*)buf.data;
    out->next = 0;
    out->sent = 0;

/* Allocate enough room for the transmitted record, which will be:
     *  5 bytes of header +
     *  encrypted contents +
     *  macLength +
     *  padding [block ciphers only] +
     *  padding length field (1 byte) [block ciphers only]
     */
    payloadSize = (uint16) (rec.contents.length + ctx->writeCipher.hash->digestSize);
    blockSize = ctx->writeCipher.symCipher->blockSize;
    if (blockSize > 0)
    {   padding = blockSize - (payloadSize % blockSize) - 1;
        payloadSize = (uint16)(payloadSize + padding + 1);
    }

   if(ctx->ssloppy)
   {
        /* in this case we need more room, for the nonce */
        nonceSize = (uint16) MAX(sizeof(uint64), ctx->writeCipher.symCipher->ivSize);
/*      payloadSize += nonceSize; decided this was wrong logic */
   )

    out->data.data = 0;
```

```
334     if ((err = SSLAllocBuffer(&out->data, 5 + payloadSize + nonceSize,
335                                                     &ctx->sysCtx)) != 0)
336         goto fail;
337
338     progress = out->data.data;
339     *(progress++) = rec.contentType;
340     progress = SSLEncodeInt(progress, rec.protocolVersion, 2);
341     progress = SSLEncodeInt(progress, payloadSize, 2);
342
343     /* Copy the contents into the output buffer */
344     memcpy(progress, rec.contents.data, (size_t) rec.contents.length);
345     payload.data = progress;
346     payload.length = rec.contents.length;
347
348     progress += rec.contents.length;
349     /* MAC immediately follows data */
350     mac.data = progress;
351     mac.length = ctx->writeCipher.hash->digestSize;
352     progress += mac.length;
353
354   if(ctx->ssloppy)
355   {
356           uint64 noncevalue;
357
358           if((err = SSLAllocBuffer(&nonce, nonceSize, &ctx->sysCtx)) != SSLNoErr)
359                   goto fail;
360           if((err = ctx->sysCtx.random(nonce, ctx->sysCtx.randomRef)) != SSLNoErr)
361                   goto fail;
362
363           memcpy(&noncevalue, nonce.data, sizeof(noncevalue));
364
365           /* MAC the data, sloppy-style */
366           if (mac.length > 0) /* Optimize away null case */
367           {
368                   secret.data = ctx->writeCipher.macSecret;
369                   secret.length = ctx->writeCipher.hash->digestSize;
370                   if ((err = ComputeMAC(rec.contentType, payload, mac, noncevalue,
371                                                           secret, &ctx->writeCipher, ctx)) != 0)
372                           goto fail;
373           }
374
375           memcpy(progress, nonce.data, nonce.length);
376           progress += nonce.length;
377
378   }
379   else
380   {
381           /* MAC the data, normal mode */
382           if (mac.length > 0) /* Optimize away null case */
383           {
384                   secret.data = ctx->writeCipher.macSecret;
385                   secret.length = ctx->writeCipher.hash->digestSize;
386                   if ((err = ComputeMAC(rec.contentType, payload, mac,
387                                                           ctx->writeCipher.sequenceNum, secret,
388                                                           &ctx->writeCipher, ctx)) != 0)
389                           goto fail;
390           }
391   }
392
393     /* Update payload to reflect encrypted data: contents, mac & padding */
394     payload.length = payloadSize;
395
396     /* Fill in the padding bytes & padding length field with the padding value; the
397      *  protocol only requires the last byte,
398      *  but filling them all in avoids leaking data
399      */
400     if (ctx->writeCipher.symCipher->blockSize > 0)
401         for (i = 1; i <= padding + 1; ++i)
402             payload.data[payload.length - i] = (uint8)padding;
403
404     /* Encrypt the data */
```

```
405     DUMP_BUFFER_NAME("cleartext data", payload);
406    if ((err = ctx->writeCipher.symCipher->encrypt(payload, payload,
407                                                                                    ctx-
       >ssloppy ? &nonce:NULL,
408                                                                                    ctx-
       >writeCipher.symCipherState,
409                                                                                    ctx))
       != 0)
410            goto fail;
411
412     DUMP_BUFFER_NAME("encrypted data", payload);
413
414     /* Enqueue the record to be written from the idle loop */
415     if (ctx->recordWriteQueue == 0)
416         ctx->recordWriteQueue = out;
417     else
418     {   queue = ctx->recordWriteQueue;
419         while (queue->next != 0)
420             queue = queue->next;
421         queue->next = out;
422     }
423
424    if(ctx->ssloppy)
425            SSLFreeBuffer(&nonce, &ctx->sysCtx);
426    else
427            /* Increment the sequence number */
428            IncrementUInt64(&ctx->writeCipher.sequenceNum);
429
430    if(freerec)
431            SSLFreeBuffer(&(rec.contents), &ctx->sysCtx);
432
433     return SSLNoErr;
434
435 fail:   /* Only for if we fail between when the WaitingRecord is allocated and when it is
       queued */
436     SSLFreeBuffer(&out->data, &ctx->sysCtx);
437     buf.data = (uint8*)out;
438     buf.length = sizeof(WaitingRecord);
439     SSLFreeBuffer(&buf, &ctx->sysCtx);
440            if(freerec)
441                    SSLFreeBuffer(&(rec.contents), &ctx->sysCtx);
442     return ERR(err);
443 }
444
445 static SSLErr
446 DecryptSSLRecord(uint8 type, SSLBuffer *payload, SSLContext *ctx)
447 {   SSLErr  err;
448     SSLBuffer   content, nonce;
449
450    if(type == SSL_application_data_ssloppy)
451    {
452            nonce.length = MAX(sizeof(uint64), ctx->readCipher.symCipher->ivSize);
453            nonce.data = payload->data + (payload->length - nonce.length);
454            payload->length -= nonce.length;
455    }
456
457     if ((ctx->readCipher.symCipher->blockSize > 0) &&
458         ((payload->length % ctx->readCipher.symCipher->blockSize) != 0))
459     {   SSLFatalSessionAlert(alert_unexpected_message, ctx);
460         return ERR(SSLProtocolErr);
461     }
462
463     /* Decrypt in place */
464     DUMP_BUFFER_NAME("encrypted data", (*payload));
465
466    if(type == SSL_application_data_ssloppy)
467    {
468            if ((err = ctx->readCipher.symCipher->decrypt(*payload, *payload, &nonce, ctx-
       >readCipher.symCipherState, ctx)) != 0)
469            {
```

```
470                     SSLFatalSessionAlert(alert_close_notify, ctx);
471                     return ERR(err);
472             }
473     }
474     else
475     {
476             if ((err = ctx->readCipher.symCipher->decrypt(*payload, *payload, NULL, ctx-
    >readCipher.symCipherState, ctx)) != 0)
477             {   SSLFatalSessionAlert(alert_close_notify, ctx);
478         return ERR(err);
479             }
480     }
481       DUMP_BUFFER_NAME("decrypted data", (*payload));
482
483 /* Locate content within decrypted payload */
484       content.data = payload->data;
485       content.length = payload->length - ctx->readCipher.hash->digestSize;
486       if (ctx->readCipher.symCipher->blockSize > 0)
487       {   /* padding can't be equal to or more than a block */
488           if (payload->data[payload->length - 1] >= ctx->readCipher.symCipher->blockSize)
489           {   SSLFatalSessionAlert(alert_unexpected_message, ctx);
490               return ERR(SSLProtocolErr);
491           }
492           content.length -= 1 + payload->data[payload->length - 1];   /* Remove block size
    padding */
493       }
494
495 /* Verify MAC on payload */
496       if (ctx->readCipher.hash->digestSize > 0)       /* Optimize away MAC for null case */
497           if(type == SSL_application_data_ssloppy)
498           {
499                   uint64 nonceNumber;
500
501                   memcpy(&nonceNumber, nonce.data, sizeof(nonceNumber));
502                   if ((err = VerifyMAC(type, content, payload->data + content.length,
503                                                           nonceNumber, ctx)) != 0)
504                   {
505                           SSLFatalSessionAlert(alert_bad_record_mac, ctx);
506                           return ERR(err);
507                   }
508           }
509           else
510           {
511                   if ((err = VerifyMAC(type, content, payload->data + content.length,
512                                                           ctx->readCipher.sequenceNum, ctx)) !=
    0)
513                   {
514                           SSLFatalSessionAlert(alert_bad_record_mac, ctx);
515                           return ERR(err);
516                   }
517           }
518
519
520       *payload = content; /* Modify payload buffer to indicate content length */
521
522       return SSLNoErr;
523 }
524
525 static uint8*
526 SSLEncodeUInt64(uint8 *p, uint64 value)
527 {   p = SSLEncodeInt(p, value.high, 4);
528       return SSLEncodeInt(p, value.low, 4);
529 }
530
531 static SSLErr
532 VerifyMAC(uint8 type, SSLBuffer data, uint8 *compareMAC, uint64 seqNo, SSLContext *ctx)
533 {   SSLErr      err;
534       uint8           macData[MAX_DIGEST_SIZE];
535       SSLBuffer       secret, mac;
536
537       secret.data = ctx->readCipher.macSecret;
```

```
538     secret.length = ctx->readCipher.hash->digestSize;
539     mac.data = macData;
540     mac.length = ctx->readCipher.hash->digestSize;
541
542     if ((err = ComputeMAC(type, data, mac, seqNo, secret,
543                     &ctx->readCipher, ctx)) != 0)
544         return ERR(err);
545
546     if ((memcmp(mac.data, compareMAC, (size_t) mac.length)) != 0)
547         return ERR(SSLProtocolErr);
548
549     return SSLNoErr;
550 }
551
552 static SSLErr
553 ComputeMAC(uint8 type, SSLBuffer data, SSLBuffer mac, uint64 seqNo, SSLBuffer secret,
554             CipherContext *cipherCtx, SSLContext *ctx)
555 {   SSLErr      err;
556     uint8           innerDigestData[MAX_DIGEST_SIZE];
557     uint8           scratchData[11], *progress;
558     SSLBuffer       digest, scratch;
559
560 #ifdef HYPER_DEBUG
561   int i;
562   fprintf(stderr, "Buffer: ");
563   for(i = 0; i < data.length; i++)
564           fprintf(stderr, "%02x ", data.data[i]);
565   fprintf(stderr, "\n");
566
567   fprintf(stderr, "sequenceno: ");
568   for(i = 0; i < sizeof(uint64); i++)
569           fprintf(stderr, "%02x ", (unsigned char) *((unsigned char *) &seqNo) + i);
570   fprintf(stderr, "\n");
571
572   fprintf(stderr, "Secret: ");
573   for(i = 0; i < secret.length; i++)
574           fprintf(stderr, "%02x ", secret.data[i]);
575   fprintf(stderr, "\n");
576 #endif
577
578     ASSERT(cipherCtx->hash->macPadSize <= MAX_MAC_PADDING);
579     ASSERT(cipherCtx->hash->digestSize <= MAX_DIGEST_SIZE);
580     ASSERT(SSLMACPad1[0] == 0x36 && SSLMACPad2[0] == 0x5C);
581
582     if(cipherCtx->digestCtx.data == NULL) {
583       if ((err = SSLAllocBuffer(&cipherCtx->digestCtx,
584                       cipherCtx->hash->contextSize, &ctx->sysCtx))
585      != 0)
586         goto exit;
587       cipherCtx->hash->create(cipherCtx->digestCtx);
588     }
589
590 /* MAC = hash( MAC_write_secret + pad_2 + hash( MAC_write_secret + pad_1 + seq_num + type +
    length + content ) ) */
591     if ((err = cipherCtx->hash->init(cipherCtx->digestCtx)) != 0)
592         goto exit;
593     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, secret)) != 0)    /* MAC secret */
594         goto exit;
595     scratch.data = SSLMACPad1;
596     scratch.length = cipherCtx->hash->macPadSize;
597     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, scratch)) != 0)   /* pad1 */
598         goto exit;
599     progress = scratchData;
600     progress = SSLEncodeUInt64(progress, seqNo);
601     *progress++ = type;
602     progress = SSLEncodeInt(progress, data.length, 2);
603     scratch.data = scratchData;
604     scratch.length = 11;
605     ASSERT(progress == scratchData+11);
606     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, scratch)) != 0)   /* sequenceNo,
    type & length */
```

```
607         goto exit;
608     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, data)) != 0)  /* content */
609         goto exit;
610     digest.data = innerDigestData;
611     digest.length = cipherCtx->hash->digestSize;
612     if ((err = cipherCtx->hash->final(cipherCtx->digestCtx, digest)) != 0) /* figure inner
      digest */
613         goto exit;
614
615     if ((err = cipherCtx->hash->init(cipherCtx->digestCtx)) != 0)
616         goto exit;
617     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, secret)) != 0)     /* MAC secret */
618         goto exit;
619     scratch.data = SSLMACPad2;
620     scratch.length = cipherCtx->hash->macPadSize;
621     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, scratch)) != 0)    /* pad2 */
622         goto exit;
623     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, digest)) != 0)     /* inner digest
     */
624         goto exit;
625     if ((err = cipherCtx->hash->final(cipherCtx->digestCtx, mac)) != 0)     /* figure the mac */
626         goto exit;
627
628     err = SSLNoErr; /* redundant, I know */
629
630 exit:
631     return ERR(err);
632 }
```

EXHIBIT D

```
/*  *********************************************************************
    File: ciphers.c

    SSL Plus: Security Integration Suite(tm)
    Version 1.1.1 -- August 11, 1997

    Copyright (c)1996, 1997 by Consensus Development Corporation
          Copyright (c)1997, 1998 by Aventail Corporation

    Portions of this software are based on SSLRef(tm) 3.0, which is
    Copyright (c)1996 by Netscape Communications Corporation. SSLRef(tm)
    was developed by Netscape Communications Corporation and Consensus
    Development Corporation.

    In order to obtain this software, your company must have signed
    either a PRODUCT EVALUATION LICENSE (a copy of which is included in
    the file "LICENSE.TXT"), or a PRODUCT DEVELOPMENT LICENSE. These
    licenses have different limitations regarding how you are allowed to
    use the software. Before retrieving (or using) this software, you
    *must* ascertain which of these licenses your company currently
    holds. Then, by retrieving (or using) this software you agree to
    abide by the particular terms of that license. If you do not agree
    to abide by the particular terms of that license, than you must
    immediately delete this software. If your company does not have a
    signed license of either kind, then you must either contact
    Consensus Development and execute a valid license before retrieving
    (or using) this software, or immediately delete this software.

    *********************************************************************

    File: ciphers.c    Data structures for handling supported ciphers

    Contains a table mapping cipherSuite values to the ciphers, MAC
    algorithms, key exchange procedures and so on that are used for that
    algorithm, in order of preference.

    ********************************************************************* */

#ifndef _CRYPTYPE_H_
#include <cryptype.h>
#endif

#ifndef _SSLCTX_H_
#include <sslctx.h>
#endif

#include <string.h>

extern SSLSymmetricCipher SSLCipherNull;
extern SSLSymmetricCipher SSLCipherDES_CBC;
extern SSLSymmetricCipher SSLCipherDES40_CBC;
extern SSLSymmetricCipher SSLCipherRC4_40;
extern SSLSymmetricCipher SSLCipherRC4_56;
extern SSLSymmetricCipher SSLCipherRC4_128;
extern SSLSymmetricCipher SSLCipher3DES_CBC;

/* Even if we don't support NULL_WITH_NULL_NULL for transport, we need a reference for startup
   */
SSLCipherSpec SSL_NULL_WITH_NULL_NULL_CipherSpec =
{   SSL_NULL_WITH_NULL_NULL,
    Exportable,
    SSL_NULL_auth,
    &SSLHashNullOpt,
    &SSLCipherNull
};

/* Disable non-exportable cipher suites to build an export only library */
#ifndef ENABLE_NONEXPORT_CIPHERS
#define ENABLE_NONEXPORT_CIPHERS 1
#endif

```

```
/* Disable exportable cipher suites to build a strong crypto only library */
#ifndef ENABLE_EXPORT_CIPHERS
#define ENABLE_EXPORT_CIPHERS 1
#endif

/* Reenable DH-anon only if you know you want to use Diffie-Hellman cipher suites:
    Enabling DH-anon leaves you open to a man-in-the-middle attack which can degrade your
     security to this level. */
#ifndef ENABLE_DH_ANON
#define ENABLE_DH_ANON 0
#endif

/* Reenable NULL encryption cipher suites only if you know for a fact you want to support
    unencrypted sessions. Unencrypted sessions do not provide data privacy and may be more
    vulnerable to attack than encrypted sessions. */
#ifndef ENABLE_NULL_CIPHERS
#define ENABLE_NULL_CIPHERS 0
#endif

#ifdef VIRGIN_SSLPLUS
/* Order by preference */
SSLCipherSpec KnownCipherSpecs[] =
{
#if ENABLE_NONEXPORT_CIPHERS
    {   SSL_RSA_WITH_3DES_EDE_CBC_SHA, NotExportable, SSL_RSA, &SSLHashSHA1, &SSLCipher3DES_CBC
    },
    {   SSL_RSA_WITH_RC4_128_SHA, NotExportable, SSL_RSA, &SSLHashSHA1, &SSLCipherRC4_128 },
    {   SSL_RSA_WITH_RC4_128_MD5, NotExportable, SSL_RSA, &SSLHashMD5, &SSLCipherRC4_128 },
    {   SSL_RSA_WITH_DES_CBC_SHA, NotExportable, SSL_RSA, &SSLHashSHA1, &SSLCipherDES_CBC },
#endif
#if ENABLE_EXPORT_CIPHERS
    {   SSL_RSA_EXPORT_WITH_RC4_40_MD5, Exportable, SSL_RSA_EXPORT, &SSLHashMD5,
   &SSLCipherRC4_40 },
    {   SSL_RSA_EXPORT_WITH_DES40_CBC_SHA, Exportable, SSL_RSA_EXPORT, &SSLHashSHA1,
   &SSLCipherDES40_CBC },
#endif
#if ENABLE_DH_ANON && ENABLE_NONEXPORT_CIPHERS
    {   SSL_DH_anon_WITH_3DES_EDE_CBC_SHA, NotExportable, SSL_DH_anon, &SSLHashSHA1,
   &SSLCipher3DES_CBC },
    {   SSL_DH_anon_WITH_RC4_128_MD5, NotExportable, SSL_DH_anon, &SSLHashMD5,
   &SSLCipherRC4_128 },
    {   SSL_DH_anon_WITH_DES_CBC_SHA, NotExportable, SSL_DH_anon, &SSLHashSHA1,
   &SSLCipherDES_CBC },
#endif
#if ENABLE_NULL_CIPHERS && ENABLE_EXPORT_CIPHERS
    {   SSL_RSA_WITH_NULL_SHA, Exportable, SSL_RSA, &SSLHashSHA1, &SSLCipherNull },
    {   SSL_RSA_WITH_NULL_MD5, Exportable, SSL_RSA, &SSLHashMD5, &SSLCipherNull }
#endif
};

int CipherSpecCount = sizeof(KnownCipherSpecs) / sizeof(SSLCipherSpec);
#endif /* VIRGIN_SSLPLUS */

SSLErr
FindCipherSpec(SSLContext *ctx, uint16 specID, SSLCipherSpec* *spec)
{
  int i;
  uint32 mask;

   *spec = 0;
   for (i = 0; i < CipherSpecCount; i++)
   {
        if (KnownCipherSpecs[i].cipherSpec == specID)
        {
                mask = (uint32) 1;
                mask <<= i;
                if(ctx->cipherspecs & mask)
                {
                        *spec = &KnownCipherSpecs[i];
                        break;
                }
```

```
135         )
136     }
137
138     if (*spec == 0)         /* Not found */
139         return SSLNegotiationErr;
140     return SSLNoErr;
141 }
142
143 SSLErr SSLDESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx);
144 SSLErr SSLDESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
      *ctx);
145 SSLErr SSLDESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
      *ctx);
146 SSLErr SSLDESFinish(void *cipherRef, SSLContext *ctx);
147 SSLErr SSLDESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob);
148 SSLErr SSLDESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob);
149
150 SSLSymmetricCipher SSLCipherDES_CBC = {
151     8,      /* Key size in bytes */
152     8,      /* Secret key size = 64 bits */
153     8,      /* IV size */
154     8,      /* Block size */
155     SSLDESInit,
156     SSLDESEncrypt,
157     SSLDESDecrypt,
158     SSLDESFinish,
159   SSLDESExport,
160   SSLDESImport
161 };
162
163 SSLSymmetricCipher SSLCipherDES40_CBC = {
164     8,      /* Key size in bytes */
165     5,      /* Secret key size = 40 bits */
166     8,      /* IV size */
167     8,      /* Block size */
168     SSLDESInit,
169     SSLDESEncrypt,
170     SSLDESDecrypt,
171     SSLDESFinish
172 };
173
174 typedef struct _DESState
175 {
176   unsigned char key[24]; /* work for 3DES and DES both */
177   unsigned char iv[8];
178   int reading; /* do we really need this? */
179   B_ALGORITHM_OBJ des;
180 } DESState;
181
182 SSLErr
183 SSLDESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx)
184 {
185   SSLBuffer                     desState;
186     B_ALGORITHM_OBJ             *des;
187     static B_ALGORITHM_METHOD  *chooser[] = { &AM_DES_CBC_ENCRYPT, &AM_DES_CBC_DECRYPT, 0 };
188     B_KEY_OBJ                   desKey;
189     ITEM                        keyData;
190     SSLErr                      err;
191     int                         rsaErr;
192   DESState *s;
193
194     if ((err = SSLAllocBuffer(&desState, sizeof(DESState), &ctx->sysCtx)) != 0)
195         return err;
196     s = (DESState *)desState.data;
197
198   memcpy(s->key, key, 8);
199   memcpy(s->iv, iv, 8);
200
201     if ((rsaErr = B_CreateAlgorithmObject(&(s->des))) != 0)
202         return SSLUnknownErr;
```

```
203     if ((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, iv)) != 0)
204         return SSLUnknownErr;
205     if ((rsaErr = B_CreateKeyObject(&desKey)) != 0)
206         return SSLUnknownErr;
207     keyData.data = key;
208     keyData.len = 8;
209     if ((rsaErr = B_SetKeyInfo(desKey, KI_DES8, key)) != 0)
210     {   B_DestroyKeyObject(&desKey);
211         return SSLUnknownErr;
212     }
213     if (cipherRef == (void**)&(ctx->writePending.symCipherState))
214     {
215           s->reading = 0;
216           if ((rsaErr = B_EncryptInit(*des, desKey, chooser, &ctx->sysCtx.yield)) != 0)
217         {
218                 B_DestroyKeyObject(&desKey);
219               return SSLUnknownErr;
220         }
221     }
222     else if (cipherRef == (void**)&(ctx->readPending.symCipherState))
223     {
224           s->reading = 1;
225           if ((rsaErr = B_DecryptInit(*des, desKey, chooser, &ctx->sysCtx.yield)) != 0)
226         {
227                 B_DestroyKeyObject(&desKey);
228               return SSLUnknownErr;
229         }
230     )
231     else
232         ASSERTMSG("Couldn't determine read/writeness");
233
234     B_DestroyKeyObject(&desKey);
235     *cipherRef = (void*)s;
236     return SSLNoErr;
237 }
238
239 SSLErr
240 SSLDESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
241 {
242     DESState *s = (DESState *) cipherRef;
243     void *subCipherRef = NULL;
244     int             rsaErr;
245     unsigned int        outputLen;
246     SSLBuffer           temp;
247     SSLErr          err;
248
249     if(cipherRef == NULL)
250           return SSLUnknownErr;
251
252     if(iv != NULL)
253     (
254           if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8,
255                                                                                   (POINTER) iv->data)) !=
      SSLNoErr)
256                 return err;
257     )
258     else
259     (
260           if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, s->iv)) != SSLNoErr)
261                 return err;
262     )
263
264     ASSERT(src.length == dest.length);
265     ASSERT(src.length % 8 == 0);
266
267     if (src.data == dest.data)
268 /* BSAFE won't let you encrypt in place */
269     (   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
270             return err;
271         memcpy(temp.data, src.data, (size_t) src.length);
272     )
```

```
273     else
274         temp = src;
275
276     if ((rsaErr = B_EncryptUpdate(s->des, dest.data, &outputLen,
277                             (unsigned int) dest.length, temp.data,
278                             (unsigned int) temp.length,
279                         (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
280     {   if (src.data == dest.data)
281             SSLFreeBuffer(&temp, &ctx->sysCtx);
282         return SSLUnknownErr;
283     }
284
285     ASSERT(outputLen == src.length);
286
287     if (src.data == dest.data)
288         SSLFreeBuffer(&temp, &ctx->sysCtx);
289
290     if (outputLen != src.length)
291         return SSLUnknownErr;
292
293   /* if not doing SSLoppy, save the IV for next time... */
294   if(iv == NULL)
295   {
296           unsigned char *buf;
297
298           if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
299                                                                   AI_DES_CBC_IV8))
300               != SSLNoErr)
301                   return err;
302
303           memcpy(s->iv, buf, sizeof(s->iv));
304   }
305
306 /* memcpy(s->iv, dest.data + dest.length - 8, 8); */
307
308     return SSLNoErr;
309 }
310
311 SSLErr
312 SSLDESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
313 {
314   DESState *s = (DESState *) cipherRef;
315     int             rsaErr;
316     unsigned int        outputLen;
317     SSLBuffer           temp;
318     SSLErr          err;
319
320   if(cipherRef == NULL)
321           return SSLUnknownErr;
322
323   if(iv != NULL)
324   {
325           if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, (POINTER) iv->data))
326               != SSLNoErr)
327                   return err;
328   }
329   else
330   {
331           if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, s->iv)) != SSLNoErr)
332                   return err;
333   }
334
335     ASSERT(src.length == dest.length);
336     ASSERT(src.length % 8 == 0);
337
338 /* memcpy(s->iv, src.data + src.length - 8, 8); */
339
340     if (src.data == dest.data)
341 /* BSAFE won't let you encrypt in place */
342     {   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
343             return err;
```

```
344         memcpy(temp.data, src.data, (size_t) src.length);
345     }
346     else
347         temp = src;
348
349     if ((rsaErr = B_DecryptUpdate(s->des, dest.data, &outputLen,
350                             (unsigned int) dest.length, temp.data,
351                             (unsigned int) temp.length,
352                         (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
353     {   if (src.data == dest.data)
354             SSLFreeBuffer(&temp, &ctx->sysCtx);
355         return SSLUnknownErr;
356     }
357
358     ASSERT(outputLen == src.length);
359
360     if (src.data == dest.data)
361         SSLFreeBuffer(&temp, &ctx->sysCtx);
362
363     if (outputLen != src.length)
364         return SSLUnknownErr;
365
366    /* if not doing SSLoppy, save the IV for next time... */
367    if(iv == NULL)
368    {
369           unsigned char *buf;
370
371           if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
372                                                                    AI_DES_CBC_IV8))
373               != SSLNoErr)
374                   return err;
375           memcpy(s->iv, buf, sizeof(s->iv));
376    }
377
378      return SSLNoErr;
379 }
380
381 SSLErr
382 SSLDESFinish(void *cipherRef, SSLContext *ctx)
383 {
384    DESState *s = (DESState *) cipherRef;
385     SSLBuffer           desState;
386     SSLErr          err;
387
388    if(cipherRef == NULL)
389           return SSLUnknownErr;
390
391     B_DestroyAlgorithmObject(&(s->des));
392
393    memset(cipherRef, 0, sizeof(DESState));
394    desState.data = (unsigned char*)cipherRef;
395     desState.length = sizeof(DESState);
396
397     err = SSLFreeBuffer(&desState, &ctx->sysCtx);
398     return err;
399 }
400
401 SSLErr SSLDESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob)
402 {
403    DESState *s = (DESState *) cipherRef;
404
405    if(cipherRef == NULL)
406           return SSLUnknownErr;
407
408    if(blob->length < (8 + 8))
409           return SSLMemoryErr;
410
411    memcpy(blob->data, s->key, 8);
412    memcpy(blob->data + 8, s->iv, 8);
413 /* memcpy(blob->data + 16, &(s->reading), sizeof(int)); */
414    blob->length = 16;
```

```
415
416     return SSLNoErr;
417 }
418
419 SSLErr SSLDESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob)
420 {
421     unsigned char *key, *iv;
422
423     if(blob == NULL)
424             return SSLUnknownErr;
425     if(blob->length < 16)
426             return SSLUnknownErr;
427
428     key = blob->data;
429     iv = blob->data + 8;
430
431     return SSLDESInit(key, iv, cipherRef, ctx);
432 }
433
434
435 SSLErr SSL3DESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx);
436 SSLErr SSL3DESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
      *ctx);
437 SSLErr SSL3DESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
      *ctx);
438 SSLErr SSL3DESFinish(void *cipherRef, SSLContext *ctx);
439 SSLErr SSL3DESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob);
440 SSLErr SSL3DESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob);
441
442 SSLSymmetricCipher SSLCipher3DES_CBC = {
443     24,     /* Key size in bytes */
444     24,     /* Secret key size = 192 bits */
445     8,      /* IV size */
446     8,      /* Block size */
447     SSL3DESInit,
448     SSL3DESEncrypt,
449     SSL3DESDecrypt,
450     SSL3DESFinish,
451   SSL3DESExport,
452   SSL3DESImport
453 };
454
455 SSLErr
456 SSL3DESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx)
457 {
458   SSLBuffer                     desState;
459   DESState *s;
460    static B_ALGORITHM_METHOD   *chooser[] = { &AM_DES_EDE3_CBC_ENCRYPT,
461
      &AM_DES_EDE3_CBC_DECRYPT, 0 };
462    B_KEY_OBJ                    desKey;
463    ITEM                         keyData;
464    SSLErr                       err;
465    int                          rsaErr;
466
467    if ((err = SSLAllocBuffer(&desState, sizeof(DESState), &ctx->sysCtx)) != 0)
468        return err;
469    s = (DESState *)desState.data;
470    if ((rsaErr = B_CreateAlgorithmObject(&(s->des))) != 0)
471        return SSLUnknownErr;
472    if ((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8, iv)) != 0)
473        return SSLUnknownErr;
474   memcpy(s->iv, iv, 8);
475
476    if ((rsaErr = B_CreateKeyObject(&desKey)) != 0)
477        return SSLUnknownErr;
478    keyData.data = key;
479    keyData.len = 24;
480    if ((rsaErr = B_SetKeyInfo(desKey, KI_24Byte, key)) != 0)
481    {
482          B_DestroyKeyObject(&desKey);
```

```
483         return SSLUnknownErr;
484     )
485     memcpy(s->key, key, 24);
486
487     if (cipherRef == (void**)&(ctx->writePending.symCipherState))
488     {
489           if ((rsaErr = B_EncryptInit(s->des, desKey, chooser,
490                                                                   &ctx->sysCtx.yield)) != 0)
491           {
492                   B_DestroyKeyObject(&desKey);
493               return SSLUnknownErr;
494           }
495     )
496     else if (cipherRef == (void**)&(ctx->readPending.symCipherState))
497     {
498           if ((rsaErr = B_DecryptInit(s->des, desKey, chooser,
499                                                                   &ctx->sysCtx.yield)) != 0)
500           {
501                   B_DestroyKeyObject(&desKey);
502               return SSLUnknownErr;
503           )
504     )
505     else
506         ASSERTMSG("Couldn't determine read/writeness");
507
508     B_DestroyKeyObject(&desKey);
509     *cipherRef = (void*)desState.data;
510     return SSLNoErr;
511 )
512
513 SSLErr
514 SSL3DESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
515 (
516     DESState *s =(DESState *) cipherRef;
517     int             rsaErr;
518     unsigned int    outputLen;
519     SSLBuffer       temp;
520     SSLErr          err;
521
522     ASSERT(src.length == dest.length);
523     ASSERT(src.length % 8 == 0);
524     if(cipherRef == NULL)
525           return SSLUnknownErr;
526
527     if(iv != NULL)
528     {
529           if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8,
530                                                                   (POINTER) iv->data)) !=
      SSLNoErr)
531                   return err;
532     }
533     else
534     {
535           if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8, s->iv)) != SSLNoErr)
536                   return err;
537     )
538
539
540     if (src.data == dest.data)
541 /* BSAFE won't let you encrypt in place */
542       (   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
543               return err;
544           memcpy(temp.data, src.data, (size_t) src.length);
545       )
546       else
547           temp = src;
548
549       if ((rsaErr = B_EncryptUpdate(s->des, dest.data, &outputLen,
550                               (unsigned int) dest.length, temp.data,
551                               (unsigned int) temp.length,
552                       (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
```

```
553     (   if (src.data == dest.data)
554             SSLFreeBuffer(&temp, &ctx->sysCtx);
555         return SSLUnknownErr;
556     }
557
558     ASSERT(outputLen == src.length);
559
560     if (src.data == dest.data)
561         SSLFreeBuffer(&temp, &ctx->sysCtx);
562
563     if (outputLen != src.length)
564         return SSLUnknownErr;
565
566   if(iv == NULL)
567   (
568           unsigned char *buf;
569
570           if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
571                                                                   AI_DES_EDE3_CBC_IV8))
572               != SSLNoErr)
573                   return err;
574           memcpy(s->iv, buf, sizeof(s->iv));
575   }
576
577 /* memcpy(s->iv, dest.data + dest.length - 8, 8); */
578
579     return SSLNoErr;
580 }
581
582 SSLErr
583 SSL3DESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
584 (
585   DESState *s = (DESState *) cipherRef;
586     int             rsaErr;
587     unsigned int        outputLen;
588     SSLBuffer           temp;
589     SSLErr          err;
590
591     ASSERT(src.length == dest.length);
592     ASSERT(src.length % 8 == 0);
593   if(cipherRef == NULL)
594           return SSLNoErr;
595
596   if(iv != NULL)
597   (
598           if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8,
599                                                                   (POINTER) iv->data)) !=
    SSLNoErr)
600                   return err;
601   }
602   else
603   (
604           if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8, s->iv)) != SSLNoErr)
605                   return err;
606   }
607
608 /* memcpy(s->iv, src.data + src.length - 8, 8);  */
609
610     if (src.data == dest.data)
611 /* BSAFE won't let you encrypt in place */
612     (   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
613             return err;
614         memcpy(temp.data, src.data, (size_t) src.length);
615     }
616     else
617         temp = src;
618
619     if ((rsaErr = B_DecryptUpdate(s->des, dest.data, &outputLen,
620                             (unsigned int) dest.length, temp.data,
621                             (unsigned int) temp.length,
622                     (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
```

```
623     {   if (src.data == dest.data)
624             SSLFreeBuffer(&temp, &ctx->sysCtx);
625         return SSLUnknownErr;
626     }
627
628    if(iv == NULL)
629    {
630            unsigned char *buf;
631
632            if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
633                                                                                    AI_DES_EDE3_CBC_IV8)) !=
       SSLNoErr)
634                    return err;
635            memcpy(s->iv, buf, sizeof(s->iv));
636    }
637
638     ASSERT(outputLen == src.length);
639
640     if (src.data == dest.data)
641         SSLFreeBuffer(&temp, &ctx->sysCtx);
642
643     if (outputLen != src.length)
644         return SSLUnknownErr;
645
646     return SSLNoErr;
647 }
648
649 SSLErr
650 SSL3DESFinish(void *cipherRef, SSLContext *ctx)
651 {
652    DESState *s = (DESState *) cipherRef;
653     SSLBuffer           desState;
654     SSLErr          err;
655
656    if(cipherRef == NULL)
657            return SSLUnknownErr;
658
659    B_DestroyAlgorithmObject(&(s->des));
660
661    memset(cipherRef, 0, sizeof(DESState));
662     desState.data = (unsigned char*)cipherRef;
663     desState.length = sizeof(DESState);
664     err = SSLFreeBuffer(&desState, &ctx->sysCtx);
665     return err;
666 }
667
668 SSLErr SSL3DESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob)
669 {
670    DESState *s = (DESState *) cipherRef;
671
672    if(cipherRef == NULL)
673            return SSLUnknownErr;
674
675    if(blob->length < (24 + 8))
676            return SSLMemoryErr;
677
678    memcpy(blob->data, s->key, 24);
679    memcpy(blob->data + 24, s->iv, 8);
680    blob->length = 32;
681
682    return SSLNoErr;
683 }
684
685 SSLErr SSL3DESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob)
686 {
687    unsigned char *key, *iv;
688
689    if(blob == NULL)
690            return SSLUnknownErr;
691    if(blob->length < 32)
692            return SSLUnknownErr;
```

```
693
694     key = blob->data;
695     iv = blob->data + 24;
696
697     return SSL3DESInit(key, iv, cipherRef, ctx);
698 }
```

EXHIBIT E

```
 0 /* Copyright Aventail Corporation 1997-2000; All Rights Reserved */
 1 /* sslenv.c is the SSL environment file; it defines functions used by
 2    the SSL module as callbacks for managing mutexes, memory, I/O,
 3    and user interaction. */
 4
 5 #include "sslmain.h"
 6 #include "sslldap.h"
 7 #include "ldapcert.h"
 8 #include <aglobal.h>
 9 #include <bsafe.h>
10 #include <pkcs.h>
```

<due to the size of this file and the small portion of it which is relevant here, we include only the single function SSLEncode>

```
970 int FAR EXPORT SSLEncode(S5Packet *ibuf, S5Packet *obuf , int flag, void *handle)
971 {
972   S5SSLHandle *ref = handle;
973   S5SSLFlowConnection *conn = &(ref->conn);
974   SSLContext *ctx = ref->ctx;
975   uint32 ilen;
976   uint32 len = ibuf ? ibuf->len : 0;
977   int ssloppy = 0;
978
979 #ifdef HYPER_DEBUG
980   int i;
981 #endif
982   SSLErr err;
983   uint32 wrtp = 0;
984
985   if (flag & S5_STATEDUMP)
986   {
987         SSLBuffer block;
988         unsigned totalSize;
989         PSSLStateDump dump = (PSSLStateDump)obuf->data;
990
991         if (obuf->len < 4096)
992         {
993               obuf->len = 4096;
994               return ENCODE_BUFFER_TOO_SMALL;
995         }
996
997         // get the SSL state
998         //
999         if ((err = SSLExportContext(ctx, &block)) != SSLNoErr)
1000        {
1001              if(GlobalUpdate)
1002                    GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_ERROR,
1003                                                IDS_SSL_EXPORTCONTEXTFAILED, err);
1004              return -1;
1005        }
1006
1007        // compute the total size of the data
1008        //
1009        totalSize = block.length + sizeof(SSLStateDump);
1010
1011        // validate the output buffer size
1012        //
1013        if (obuf->len < totalSize)
1014        {
1015              obuf->len = totalSize;
1016              SSLFreeBuffer(&block, &ctx->sysCtx);
1017              return ENCODE_BUFFER_TOO_SMALL;
1018        }
1019
1020        // put the SSLStateDump structure at the beginning of the output buffer
1021        //
1022        dump->SSLContext = ctx;
1023        dump->ContextSize = sizeof(SSLContext);
1024        dump->SSLState.data = (uint8 *)(dump+1);
```

```
1025            dump->SSLState.length = block.length;
1026
1027            // copy the SSL state to the output buffer
1028            //
1029            memcpy(dump->SSLState.data, block.data, block.length);
1030            obuf->len = totalSize;
1031            SSLFreeBuffer(&block, &ctx->sysCtx);
1032
1033            if (GlobalUpdate)
1034                    GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1035                                                    IDS_SSL_EXPORTEDCONTEXT, obuf->len);
1036
1037            return 0;
1038    }
1039
1040    if(ref->endtime > 0)
1041            if(time((time_t *) NULL) >= ref->endtime) {
1042                    if(GlobalUpdate)
1043                            GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1044                                                    IDS_SSL_LIFETIMEEXCEEDED);

1045                    return -1;
1046            }
1047
1048    SSLGetWritePendingSize(ctx, &wrtp);
1049 #ifdef HYPER_DEBUG
1050    if(wrtp)
1051            if(GlobalUpdate)
1052                    GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1053                                            IDS_SSL_BYTESPENDINGWRITE, wrtp);
1054 #endif
1055
1056    if(flag & S5_DATAGRAM)
1057            if(ref->ssloppy)
1058            {
1059                    if((rt = SSLSetSloppyMode(ctx, 1)) != SSLNoErr)
1060                    {
1061                            if(GlobalUpdate)
1062                                    GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_WARNING,
1063                                                            IDS_SSL_SSLOPPYMODEFAILED, rt);
1064                            return -1;
1065                    }
1066                    ssloppy = 1;
1067            }
1068            else
1069            {
1070 /* UDP naked, baby! */
1071 #ifdef HYPER_DEBUG
1072                    if(GlobalUpdate)
1073                            GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1074                                                    IDS_SSL_GOTDATAGRAM, flag, ibuf->len);
1075 #endif
1076
1077 #ifdef OPTIMIZE_UDP_NAKED
1078                    /* this is much cleaner and faster, but causes inconsistency in the
1079                       API from the caller.  Sigh. */
1080                    *obuf = *ibuf;
1081 #else
1082 #ifdef WIN32
1083                    obuf->data = HeapAlloc(GetProcessHeap(), 0, ibuf->len);
1084 #else
1085                    obuf->data = malloc(ibuf->len);
1086 #endif
1087                    if(obuf->data == NULL) {
1088 #ifdef HYPER_DEBUG
1089                            FPRINTF(stderr, _T("Returning error, buf data is null in
       obufdata\n"));
1090 #endif
1091                            return SSLMemoryErr;
1092                    }
1093                    memcpy(obuf->data, ibuf->data, ibuf->len);
```

```
1094                    obuf->len = ibuf->len;
1095 #endif
1096                    return ibuf->len;
1097            }
1098
1099
1100
1101    if(flag & S5_ENCODE) {
1102
1103 #ifdef HYPER_DEBUG
1104            if(GlobalUpdate)
1105                    GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1106                                                    IDS_SSL_ENCODINGBYTES, ibuf->len);
1107 #ifndef AUTOSOCKS
1108            for(i = 0; i < ibuf->len; i++)
1109                    FPRINTF(stderr, _T("%02x "), ibuf->data[i]);
1110            FPRINTF(stderr, _T("\n"));
1111 #endif
1112 #endif
1113
1114 #define SSL_MAX_ENCODE_SIZE 4096
1115
1116 #ifdef SSL_MAX_ENCODE_SIZE
1117            if(ibuf->len > SSL_MAX_ENCODE_SIZE) {
1118                    conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1119                                                                    IDS_SSL_MAXENCODESIZEEXCEEDED,
1120                                                                    ibuf->len, SSL_MAX_ENCODE_SIZE);
1121                    ibuf->len = SSL_MAX_ENCODE_SIZE;
1122            }
1123 #endif
1124
1125            if(obuf->data != NULL) {
1126                    if(obuf->len < (int) (ibuf->len + SSL_HEADLEN + 64 + wrtp)) {
1127                            conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_DEBUG,
1128                                                                            IDS_SSL_BUFFERTOOSHORT,
       obuf->len,
1129                                                                            (ibuf->len + SSL_HEADLEN
       + 64 + wrtp));
1130                            obuf->len = (int) ibuf->len + SSL_HEADLEN + 64 + wrtp;
1131                            if(ssloppy) SSLSetSloppyMode(ctx, 0);
1132                            return ENCODE_BUFFER_TOO_SMALL;
1133                    }
1134                    conn->writebuffer.data = obuf->data;
1135                    conn->writebuffer.len = obuf->len;
1136                    conn->writebuffer.off = SSL_HEADLEN;
1137                    conn->writeflag = SSL_FLOW_WRITE_NOMAKEBUF;
1138            } else
1139                    conn->writeflag = 0;
1140
1141            ilen = (uint32) ibuf->len;
1142            if((err = SSLWrite(ibuf->data, &ilen, ctx))) {
1143                    conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_ERROR,
1144                                                                    IDS_SSL_WRITEERROR,err);
1145                    if(ssloppy) SSLSetSloppyMode(ctx, 0);
1146                    return -1;
1147            }
1148
1149            if(conn->writebuffer.off > 0xFFFF) {
1150                    conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_ERROR,
1151                                                                    IDS_SSL_PACKETTOOBIG,
1152                                                                    conn->writebuffer.off);
1153                    if(ssloppy) SSLSetSloppyMode(ctx, 0);
1154                    return -1;
1155            }
1156
1157            if(obuf->data != NULL) {
1158                    /* Here we shift the semantics of writebuffer; off now points
1159                       to the beginning of the data, and len points to the end of
1160                       the data, not the length of the buffer, which we no longer
1161                       need to know since we don't be depositing anything new in it */
1162                    obuf->len = conn->writebuffer.off;
```

```
1163                    conn->writebuffer.len = conn->writebuffer.off;
1164                    conn->writebuffer.off = SSL_HEADLEN;
1165            } else {
1166                    if(conn->writebuffer.off == SSL_HEADLEN)
1167                            /* Wow!  Some thoughtful soul in SSLFlowWrite has left us a 4
1168                               byte offset in the writebuffer so we can insert our header
1169                               without needing to malloc a new buffer and memcpy into it! */
1170                            obuf->data = conn->writebuffer.data;
1171                    else {
1172 #ifndef _WINDOWS
1173                            obuf->data = malloc(conn->writebuffer.len + SSL_HEADLEN);
1174 #else
1175 #ifdef WIN32
1176                            obuf->data = HeapAlloc(GetProcessHeap(), 0,
1177                                                                    conn->writebuffer.len +
      SSL_HEADLEN);
1178 #endif
1179 #endif
1180                            if(obuf->data == NULL) {
1181                                    conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,
1182
              S5_LOG_ERROR,IDS_SSL_MALLOCFAILED);
1183                                    if(ssloppy) SSLSetSloppyMode(ctx, 0);
1184                                    return -1;
1185                            }
1186
1187                            memcpy(obuf->data + SSL_HEADLEN,
1188                                    conn->writebuffer.data + conn->writebuffer.off,
1189                                    conn->writebuffer.len - conn->writebuffer.off);
1190 #ifdef WIN32
1191                            HeapFree(GetProcessHeap(), 0, conn->writebuffer.data);
1192 #else
1193                            free(conn->writebuffer.data);
1194 #endif
1195                    }
1196
1197                    obuf->len = (int) (conn->writebuffer.len -
1198                                                    conn->writebuffer.off + SSL_HEADLEN);
1199            }
1200
1201            obuf->data[0] = SSL_HEADVERSION;
1202            obuf->data[1] = conn->state;
1203            obuf->data[2] = (uint8) ((conn->writebuffer.len - conn->writebuffer.off)
1204                                                            >> 8);
1205            obuf->data[3] = (uint8) ((conn->writebuffer.len - conn->writebuffer.off)
1206                                                            & 0xFF);
1207            conn->writebuffer.data = NULL;
1208            conn->writebuffer.len = 0;
1209            conn->writebuffer.off = 0;
1210            if(GlobalUpdate)
1211                    GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1212                                            IDS_SSL_ENCODERETURNING, obuf->len, ilen);
1213 #ifdef HYPER_DEBUG
1214 #ifndef AUTOSOCKS
1215            for(i = 0; i < obuf->len; i++)
1216                    FPRINTF(stderr, _T("%02x "), obuf->data[i]);
1217            FPRINTF(stderr, _T("\n"));
1218 #endif
1219 #endif
1220            if(ssloppy) SSLSetSloppyMode(ctx, 0);
1221            return (int) ilen;
1222    }
1223
1224    /* we must be decoding, instead of encoding.. */
1225 #ifdef HYPER_DEBUG
1226    if(GlobalUpdate)
1227            GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1228                                    IDS_SSL_DECODINGBYTES, ibuf->len);
1229    if(GlobalUpdate)
1230            GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1231                                    IDS_SSL_READBUFFERCOMINGIN,
```

```
1232                                    conn->readbuffer.len - conn->readbuffer.off);
1233 #ifndef AUTOSOCKS
1234   for(i = 0; i < ibuf->len; i++)
1235           FPRINTF(stderr, _T("%02x "), ibuf->data[i]);
1236   FPRINTF(stderr, _T("\n"));
1237 #endif
1238 #endif
1239
1240   if(ibuf->len < SSL_HEADLEN)
1241   {
1242           conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_ERROR,
1243                                                   IDS_SSL_DECODEINCOMPLETEPACKET);
1244           if(ssloppy) SSLSetSloppyMode(ctx, 0);
1245           return -1;
1246   )
1247
1248   if(ibuf->data[0] != SSL_HEADVERSION) {
1249           conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_ERROR,
1250                                                   IDS_SSL_HEADERVERSIONMISMATCH,
1251                                                   SSL_HEADVERSION, ibuf->data[0]);
1252           if(ssloppy) SSLSetSloppyMode(ctx, 0);
1253           return -1;
1254   )
1255   if(ibuf->data[1] != conn->state) {
1256           conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_ERROR,
1257                                                   IDS_SSL_HEADERSTATEMISMATCH,
1258                                                   conn->state, ibuf->data[1]);
1259           if(ssloppy) SSLSetSloppyMode(ctx, 0);
1260           return -1;
1261   }
1262
1263   ilen = ((uint8) ibuf->data[2]) << 8;
1264   ilen |= (uint8) ibuf->data[3];
1265   ilen += SSL_HEADLEN; /* we must include the header in the length because
1266                                           no man is an ilen.  Er, because it's the length
1267                                           of the whole record, including the header. */
1268
1269 #ifdef HYPER_DEBUG
1270   if(GlobalUpdate)
1271           GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1272                                   IDS_SSL_PACKETSIZE, ilen);
1273 #endif
1274
1275   if(ibuf->len < (int) (ilen)) {
1276           conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_ERROR,
1277                                                   IDS_SSL_DECODEINCOMPLETEPACKET);
1278           if(ssloppy) SSLSetSloppyMode(ctx, 0);
1279           return -1;
1280   )
1281
1282 #if 0
1283   if(ibuf->len > (int) (ilen)) (
1284     conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_ERROR,
1285                                           IDS_SSL_DECODEOVERFULLPACKET);
1286   if(ssloppy) SSLSetSloppyMode(ctx, 0);
1287     return -1;
1288   )
1289 #endif
1290
1291   SSLGetReadPendingSize(ctx, &wrtp); /* this should be zero, but seems to
1292                                   not always be, so we be safe.. */
1293
1294 #if 0
1295   /* we need to choose the size of the obuf here; since SSL adds some
1296      boundary information the size of the input should be big enough.
1297      If SSL+ starts to support compression this assumption will have
1298      to change. */
1299   len = conn->readbuffer.len - conn->readbuffer.off + wrtp + ilen;
1300 #else
1301   /* OK, so we decided to change it.  Now we know the record can't
1302           be larger than 16K. */
```

```
1303 /*  len = conn->readbuffer.len - conn->readbuffer.off + wrtp + 16384;  */
1304   /* Hmm.. the above seems to cause telnet to studder, while a fixed 32k
1305            size fixes it, so 32k it shall be... */
1306   len = 32767;
1307 #endif
1308   if(obuf->data != NULL) {
1309     if(obuf->len < (int) len) {
1310       if(GlobalUpdate)
1311             GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_DEBUG,
1312                                        IDS_SSL_BUFFERTOOSMALL, obuf->len, len);
1313       obuf->len = (int) len;
1314     if(ssloppy) SSLSetSloppyMode(ctx, 0);
1315       return ENCODE_BUFFER_TOO_SMALL;
1316     }
1317   } else {
1318 #ifndef _WINDOWS
1319         obuf->data = (unsigned char *) malloc(len);
1320 #else
1321 #ifdef WIN32
1322         obuf->data = HeapAlloc(GetProcessHeap(), 0, len);
1323 #endif
1324 #endif
1325   }
1326 #if 0
1327   /* must read all the data we can.. */
1328   len = ilen + conn->readbuffer.len;
1329 #endif
1330
1331   if (conn->readbuffer.data == NULL) {
1332 /*     conn->readbuffer.data = malloc((size_t) (len - SSL_HEADLEN)); */
1333                 /* Try to re-use the input buffer instead of needing to create
1334                            a new one and memcpy into it.  readflag lets us know we did
1335                            this so we don't try to change or free the space later */
1336     conn->readbuffer.data = ibuf->data;
1337     conn->readbuffer.len = ilen;
1338     conn->readbuffer.off = SSL_HEADLEN;
1339     conn->readflag = SSL_FLOW_READ_NOOWNBUF;
1340   } else {
1341     conn->readbuffer.data = realloc(conn->readbuffer.data,
1342                                     (size_t) (len - SSL_HEADLEN));
1343     conn->readflag = 0;
1344     if(conn->readbuffer.data == NULL) {
1345       conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_ERROR,
1346                                                       IDS_SSL_REALLOCFAILED);
1347     if(ssloppy) SSLSetSloppyMode(ctx, 0);
1348       return -1;
1349     }
1350     memcpy(conn->readbuffer.data + conn->readbuffer.len,
1351       ibuf->data + SSL_HEADLEN, (size_t) (ilen - SSL_HEADLEN));
1352     conn->readbuffer.len += ilen - SSL_HEADLEN;
1353   }
1354
1355   if((err = SSLRead((void *) obuf->data, &len, ctx)) &&
1356      (err != SSLWouldBlockErr)) {
1357     conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_ERROR,
1358                                                     IDS_SSL_READERROR, err);
1359   if(ssloppy) SSLSetSloppyMode(ctx, 0);
1360     return -1;
1361   }
1362   obuf->len = (int) len;
1363 #ifdef HYPER_DEBUG
1364   if(GlobalUpdate)
1365     GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1366                                 IDS_SSL_ENCODERETURNINGBYTES,
1367                                 ilen, len);
1368   if(GlobalUpdate)
1369     GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1370                                 IDS_SSL_READBUFFERGOINGOUT,
1371                                 conn->readbuffer.len - conn->readbuffer.off);
1372 #ifndef AUTOSOCKS
1373   for(i = 0; i < len; i++)
```

```
1374            FPRINTF(stderr, _T("%02x "), obuf->data[i]);
1375    FPRINTF(stderr, _T("\n"));
1376 #endif
1377 #endif
1378    if(conn->readflag & SSL_FLOW_READ_NOOWNBUF)
1379            if (conn->readbuffer.off < conn->readbuffer.len) {
1380                    BYTE *t;
1381
1382                    if(GlobalUpdate)
1383                            GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_WARNING,
1384                                                    IDS_SSL_ENCODELEAVINGDATA,
1385                                                    conn->readbuffer.len - conn->readbuffer.off);
1386                    t = malloc(conn->readbuffer.len - conn->readbuffer.off);
1387                    memcpy(t, conn->readbuffer.data + conn->readbuffer.off,
1388                              conn->readbuffer.len - conn->readbuffer.off);
1389                    conn->readbuffer.data = t;
1390                    conn->readbuffer.len = conn->readbuffer.len - conn->readbuffer.off;
1391                    conn->readbuffer.off = 0;
1392            } else {
1393                    conn->readbuffer.data = NULL;
1394                    conn->readbuffer.off = 0;
1395                    conn->readbuffer.len = 0;
1396            }
1397    if(ssloppy) SSLSetSloppyMode(ctx, 0);
1398    return (int) ilen;
1399 }
```